

DIGITAL SIGNAL PROCESSING

MINI-PROJECT

IMAGE ENHANCEMENT

EE21B021

JACOB J PANICKER

CONTENTS

Sl.no.	Content	Page no.
1	Introduction	3
2	Brief Summary	3
3	Theory	4-7
4	(Mini-Project Topic) Noise Reduction	8-10
5	Code Implementation	11-13
6	Extra coding done in MATLAB	14-15
7	Results	16-20
8	References	21

INTRODUCTION

Image enhancement encompasses a multidisciplinary field utilizing techniques from signal processing, computer vision, and human perception to improve the quality and interpretability of digital images. This involves preprocessing methods like noise reduction and contrast enhancement, as well as spatial and frequency domain operations such as convolution and Fourier analysis. Color enhancement is applied in the case of color images, and machine learning approaches, including deep neural networks, are increasingly employed. Image enhancement finds applications in diverse fields such as medical imaging, remote sensing, and astronomy, aiming to enhance visual quality for improved analysis and interpretation. Evaluation metrics like signal-to-noise ratio and structural similarity index are used to assess the effectiveness of these techniques. Ultimately, image enhancement in academia is a systematic application of mathematical and computational methods to elevate the visual utility of digital images across various disciplines.

BRIEF SUMMARY

This mini-project aims to demonstrate the impact of each convolution kernel on picture quality by implementing filter functionality in Python using library packages like OpenCV, matplotlib and NumPy. Eliminating background noise from the source picture is also part of the goal.

THEORY

Image enhancement typically refers to the process of improving the visual quality of an image. There are various techniques and tools available for image enhancement, and the choice of method depends on the specific goals and characteristics of the image.

Below are few of the techniques/methods: -

- Brightness and Contrast Adjustment
- Color Correction
- Sharpness Enhancement
- Noise Reduction
- Image Resizing
- Histogram Equalization
- Saturation Adjustment
- HDR (High Dynamic Range)
- Image Filtering
- Dehazing

In this mini-project, we'll be focusing on the noise reduction aspect, as well as grazing over the other techniques, by using filters and implementing a code in Python using Python libraries.

SHARPENING

Digital photographs may be enhanced using a technique called image sharpening to **make them seem more detailed**. The process of sharpening makes an image's edges stand out more. Images with low quality at the margins are drab. Background and edges are almost indistinguishable. In contrast, a sharpened picture is one where the viewer can make out the edges. (Kernel given below)

Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
----------------	---

EDGE DETECTION

By using edge detection, viewers are able to scrutinize an image's characteristics for a notable shift in grayscale. With this texture, we can see where one area of the picture ends and another begins. It maintains a picture's structural integrity

while decreasing the quantity of data in the image. This is a popular technique for **deep learning** feature extraction from images.

Ridge or edge detection	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$

BOX BLUR

Computer graphics and image processing often make use of **box blur**, a straightforward method for blurring images. In it, inside a predetermined rectangular region called the "box," the average color value of each pixel is calculated by adding up the values of all the pixels immediately around it. The method softens and reduces the appearance of distinct characteristics by reducing high-frequency noise and smoothing out small details. The blur effect is proportional to the box's size; bigger boxes result in more noticeable blurring. Since it **doesn't need a lot of processing power**, box blur is a go-to for real-time uses like video games and editing multimedia, where simple and fast blurring is needed.

Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
--------------------------	---

MEDIAN BLUR

As discussed before, there are kernels for each purpose such as sharpening, blurring and edge detection, but median blur doesn't. Instead, it follows a mathematical procedure.

Median blur includes taking the average value of the neighboring pixel intensities and replaces the pixel which is getting interfered by the noise. This is very much effective against salt and pepper noise.

CONTRAST ADJUSTMENT

One of the most fundamental techniques in digital image processing, contrast adjustment **raises the brightness differences** between different areas of a picture. Visuals are brought to life and details are brought to light with the help of contrast adjustment, which emphasizes variations in brightness or colour. Equalization, histogram stretching, linear and non-linear contrast modifications, and the expansion or contraction of pixel value ranges are all used to bring intensity levels into harmony. With the use of contrast adjustment, digital images in fields such as photography, medical imaging, computer vision, and adaptive satellite imagery may be better understood and analysed, and visual information can be improved.

HISTOGRAM EQUALIZATION

Method for processing images by distributing pixel values across a broader range, histogram equalization enhances contrast. This technique **improves the dynamic range of pixel values** by making the histogram more consistent. The cumulative distribution function (CDF) of the image's pixel intensities is computed and pixel values are remapped via histogram equalization. This evens out the intensities, which widens the histogram and makes fine details more visible, particularly in areas with low contrast. Global contrast is improved by histogram equalization, while local contrast is addressed by adaptive variations. Careful application is required to remove artifacts and maintain image quality, but its effectiveness makes it valuable in medical imaging, computer vision, and remote sensing.

BRIGHTNESS ADJUSTMENT

Brightness alteration **alters the intensity or luminance of an image**. The brightness of a picture is adjusted uniformly across all pixel values. Optimizing visual perception and achieving scene lighting balance are both helped by this modification, which may repair images that are under- or overexposed. Photo editing, video editing, and computer vision all rely on brightness and contrast adjustments to produce higher-quality digital images.

Original Image and Enhanced Images using imadjust, histeq, and adapthisteq

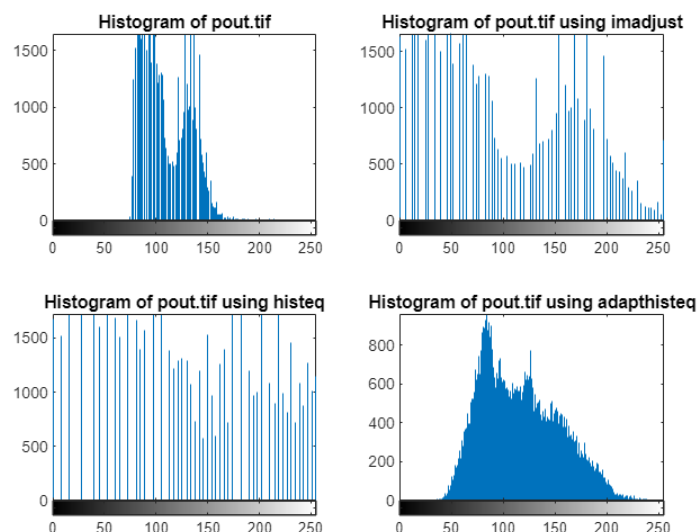


(Starting from left) The first image is the original (pout.tif).

The **second image** is the image with **increased contrast**. This is done by mapping the values of the input intensity image to new values such that, by default, 1% of the data is saturated at low and high intensities of the input data.

The **third image** is the result of **histogram equalization**. It enhances the contrast of images by transforming the values in an intensity image so that the histogram of the output image approximately matches a specified histogram (uniform distribution by default).

The **fourth image** is the result of **contrast-limited adaptive histogram equalization**. Unlike normal histogram equalization, it operates on small data regions (tiles) rather than the entire image. Each tile's contrast is enhanced so that the histogram of each output region approximately matches the specified histogram (uniform distribution by default). The contrast enhancement can be limited in order to avoid amplifying the noise which might be present in the image.



NOISE REDUCTION

Unwanted changes in pixel values that do not belong in the scene are called noise in pictures. Sensor limitations, transmission interference, and environmental factors are among the causes of image noise. Improving an image's quality and sharpness by reducing or eliminating these undesired changes is what noise reduction is all about.

There can be different types of noises, but in terms of being more studied upon, there are broadly two: -

1. Gaussian Noise: This type of noise is characterized by random variations that follow a Gaussian (normal) distribution. It appears as random speckles and can affect both the brightness and colour of individual pixels.
2. Salt and Pepper Noise: This type of noise introduces random occurrences of very bright (salt) or very dark (pepper) pixels. It simulates the appearance of random, isolated outliers in the image.

The objective of noise reduction techniques is to maintain the key elements of a picture while simultaneously smoothing out or filtering out the changes that are not desired.

Mentioned below are two techniques: -

1. Gaussian Blur: This method involves applying a convolution operation with a Gaussian kernel to the image. The convolution blurs the image, reducing high-frequency noise while preserving the overall structure. The size of the Gaussian kernel determines the extent of smoothing.
2. Median Filtering: This method replaces each pixel value with the median value in its neighborhood. Median filtering is particularly effective at removing salt and pepper noise, as it replaces outliers with more typical pixel values.

MATHEMATICAL ANALYSIS

Generally, an image is considered as a **matrix arrangement of pixels**, so each pixel has co-ordinates of the form (row number, column number). Depending on the image, it can have more than one **channels**. A channel refers to a grayscale image component representing the intensity of light for a specific colour. In a standard colour image, we often have three channels corresponding to the primary colours: red, green, and blue. In this project, we'll be focusing on **images with only one channel**, i.e., grayscale images.

Each pixels have a value which can range from 0 to 255, where these values indicate the intensity of a pixel, so 0 indicates minimum intensity (black) and 255 indicates maximum intensity (white).

In order to make variations to the values of these pixels, we make use of a function which is called the **kernel**. A kernel, convolution matrix, or mask is a small matrix used for blurring, sharpening, embossing, edge detection, and more. This is accomplished by doing a **convolution between the kernel and an image**.

Convolution is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other. It is the process of adding each element of the image to its local neighbours, weighted by the kernel.

When doing the convolution, the kernel is to be placed on top of the convolution matrix such that the centre element of the kernel coincides with the pixel of the image which we want to undergo the change. When doing so, a problem may arise, due to the fact that the kernel might go out of bounds of the image. In such cases, **edge handling** is done. Some of them are noted below: -

Extend: The nearest border pixels are conceptually extended as far as necessary to provide values for the convolution. Corner pixels are extended in 90° wedges. Other edge pixels are extended in lines.

Wrap: The image is conceptually wrapped (or tiled) and values are taken from the opposite edge or corner.

Mirror: The image is conceptually mirrored at the edges. For example, attempting to read a pixel 3 units outside an edge reads one 3 units inside the edge instead.

Constant: Use constant value for pixels outside of image. Usually black or sometimes grey is used. Generally, this depends on application.

For simplicity, the technique used in this project is the **constant edge handling**, padding with zeroes.

$$\begin{array}{ccc}
 \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} & * & \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 3 & 4 & 1 \\ 1 & 2 & 4 & 3 & 3 \\ 1 & 2 & 3 & 4 & 1 \\ 1 & 3 & 3 & 1 & 1 \\ 3 & 3 & 1 & 1 & 0 \end{pmatrix} \\
 I & K & I * K
 \end{array}$$

$$\begin{array}{ccc}
 \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} & * & \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 3 & 4 & 1 \\ 1 & 2 & 4 & 3 & 3 \\ 1 & 2 & 3 & 4 & 1 \\ 1 & 3 & 3 & 1 & 1 \\ 3 & 3 & 1 & 1 & 0 \end{pmatrix} \\
 I & K & I * K
 \end{array}$$

$$\begin{array}{ccc}
 \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} & * & \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 3 & 4 & 1 \\ 1 & 2 & 4 & 3 & 3 \\ 1 & 2 & 3 & 4 & 1 \\ 1 & 3 & 3 & 1 & 1 \\ 3 & 3 & 1 & 1 & 0 \end{pmatrix} \\
 I & K & I * K
 \end{array}$$

Convolution of an image matrix with kernel
(No edge handling)

CODE IMPLEMENTATION

```
def convert_image_matrix(img_name):
    src = cv2.imread(img_name)
    img = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
    name, ext = img_name.split('.')
    plt.imshow(str(name + '_gray.' + ext), img, cmap='gray')

    gray_img = cv2.imread(str(name + '_gray.' + ext), 0)
    gim_shape = gray_img.shape
    gim_mat = []
    for i in range(0, gim_shape[0]):
        row = []
        for j in range(0, gim_shape[1]):
            pixel = gray_img.item(i, j)
            row.append(pixel)
        gim_mat.append(row)
    gim_mat = np.array(gim_mat)
    return gim_mat
```

The input picture is first grey-scaled and then transformed into a pixel matrix where each pixel's value ranges from 0 to 255 representing the intensity of that pixel at that location.

However, the conversion is only necessary for colour photos and those pictures already in the grayscale mode can be directly converted into the image matrix.

```
def get_sub_matrices(orig_matrix, kernel_size):
    width = len(orig_matrix[0])
    height = len(orig_matrix)
    if kernel_size[0] == kernel_size[1]:
        if kernel_size[0] > 2:
            orig_matrix = np.pad(orig_matrix, kernel_size[0] - 2, mode='constant')
        else: pass
    else: pass

    giant_matrix = []
    for i in range(0, height - kernel_size[1] + 1):
        for j in range(0, width - kernel_size[0] + 1):
            giant_matrix.append(
                [
                    [orig_matrix[col][row] for row in range(j, j + kernel_size[0])]
                    for col in range(i, i + kernel_size[1])
                ]
            )
    img_sampling = np.array(giant_matrix)
    return img_sampling
```

Just as discussed before, the problem of edge handling arises in this situation.

The amount of edge handling required also depends on the kernel size, so care must be taken there too. Most of the kernels are of the size 3, but size 5 also exists.

(Size N means NxN matrix)

At the end of this stage, we'll get an output matrix which is then further used for convolution with the kernel function.

```
def get_transformed_matrix(matrix_sampling, kernel_filter):
    transform_mat = []
    for each_mat in matrix_sampling:
        transform_mat.append(
            np.sum(np.multiply(each_mat, kernel_filter))
        )
    reshape_val = int(math.sqrt(matrix_sampling.shape[0]))
    transform_mat = np.array(transform_mat).reshape(reshape_val, reshape_val)
    return transform_mat
```

Convolution of the image matrix with the kernel matrix. Note that the orientation of the kernel with respect to the transformed image matrix should be like what was discussed earlier.

```
def original_VS_convoluted(img_name, kernel_name, convoluted_matrix):
    name, ext = img_name.split('.')
    cv2.imwrite(str(name + '_' + kernel_name + '.' + ext), convoluted_matrix)
    orig = cv2.imread(str(name + '_gray.' + ext))
    conv = cv2.imread(str(name + '_' + kernel_name + '.' + ext))

    fig = plt.figure(figsize=(16, 25))
    ax1 = fig.add_subplot(2,2,1)
    ax1.axis("off")
    ax1.title.set_text('Original')
    ax1.imshow(orig)
    ax2 = fig.add_subplot(2,2,2)
    ax2.axis("off")
    ax2.title.set_text(str(kernel_name).title())
    ax2.imshow(conv)
    return True
```

The final segment deals with the comparison of the original or grayscale version of the input picture with its final filtered output. The final and the grayscale image will be saved in the directory the program was saved in.

To summarise, the steps involved in implementing the code: -

1. Conversion of the image into a greyscale version
2. Taking care of edge handling
3. Performing convolution on the output image after step 2 with the convolution kernel
4. Printing out the final filtered picture and compare it with the greyscale version of the original image

Now, as mentioned before, if trying to implement Median Blur, we see that there is no specific kernel for that. Instead, we have an algorithm or mathematical procedure.

Median blur includes taking the average value of the neighboring pixel intensities and replaces the pixel which is getting interfered by the noise. This is very much effective against salt and pepper noise.

```
def median_blur_custom(image, kernel_size):
    height, width = image.shape
    half_kernel = kernel_size // 2
    result = np.zeros((height, width), dtype=np.uint8)

    for i in range(half_kernel, height - half_kernel):
        for j in range(half_kernel, width - half_kernel):
            # Extract the local neighborhood
            neighborhood = image[i - half_kernel : i + half_kernel + 1, j - half_kernel : j + half_kernel + 1]

            # Calculate the median value and assign it to the result image
            result[i, j] = np.median(neighborhood)

    return result
```

Also, given below is the pseudo-code for convolution: -

```
for each image row in input image:
    for each pixel in image row:

        set accumulator to zero

        for each kernel row in kernel:
            for each element in kernel row:

                if element position corresponding* to pixel position then
                    multiply element value corresponding* to pixel value
                    add result to accumulator
                endif

        set output image pixel to accumulator
```

MATLAB Codes (Extras)

```
pout = imread("pout.tif");  
pout_imadjust = imadjust(pout);  
pout_histeq = histeq(pout);  
pout_adapthisteq = adapthisteq(pout);
```

(In order)

Contrast adjustment

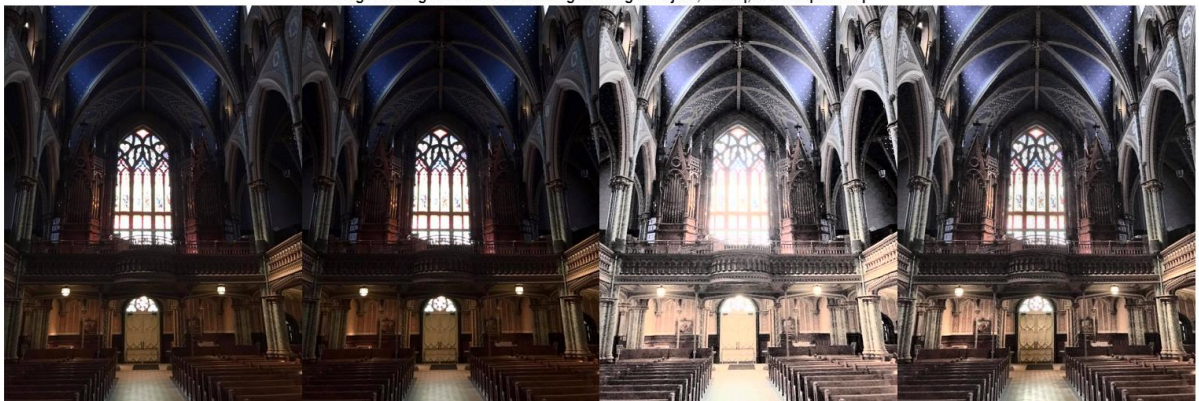
Histogram equalization

Contrast-limited adaptive histogram equalization

```
shadow_imadjust = shadow_lab;  
shadow_imadjust(:,:,1) = imadjust(L) * max_luminosity;  
shadow_imadjust = lab2rgb(shadow_imadjust);  
  
shadow_histeq = shadow_lab;  
shadow_histeq(:,:,1) = histeq(L) * max_luminosity;  
shadow_histeq = lab2rgb(shadow_histeq);  
  
shadow_adapthisteq = shadow_lab;  
shadow_adapthisteq(:,:,1) = adapthisteq(L) * max_luminosity;  
shadow_adapthisteq = lab2rgb(shadow_adapthisteq);
```

Colour Enhancement

Original Image and Enhanced Images using imadjust, histeq, and adapthisteq





Contrast enhancement of colour images is typically done by converting the image to a colour space that has image luminosity as one of its components, such as the **L*a*b* colour space**. Contrast adjustment is performed on the luminosity layer **L*** only, and then the image is converted back to the RGB colour space. Manipulating luminosity affects the

intensity of the pixels, while preserving the original colours.

Read an image with poor contrast into the workspace. Then, convert the image from the RGB colour space to the $L^*a^*b^*$ colour space.

The **CIELAB colour space**, also referred to as $L^*a^*b^*$, is a colour space defined by the International Commission on Illumination (abbreviated CIE) in 1976. It expresses colour as three values: L^* for perceptual lightness and a^* and b^* for the four unique colours of human vision: red, green, blue and yellow.

RESULTS

<p>Original</p>	
<p>Gaussian Blur</p>	

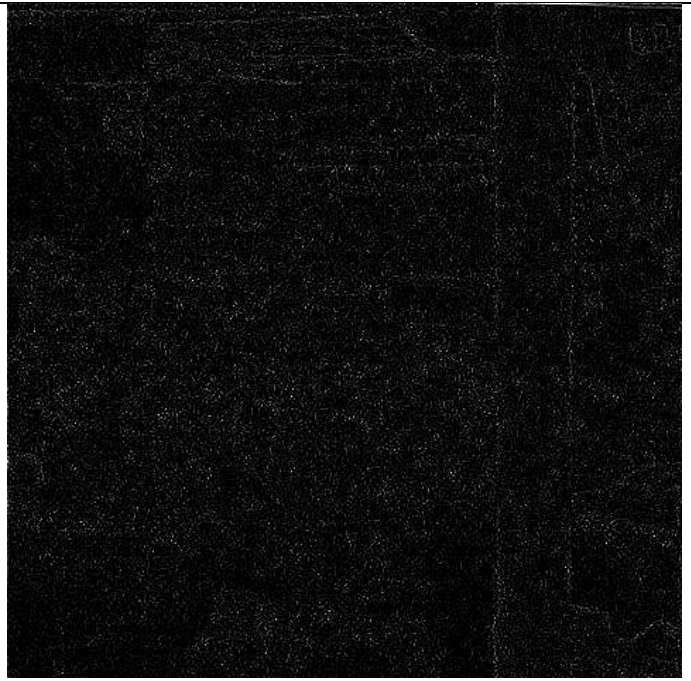
Median Blur



Sharpening



Edge Detection



Box Blur



Extra images



Original



Sharpened



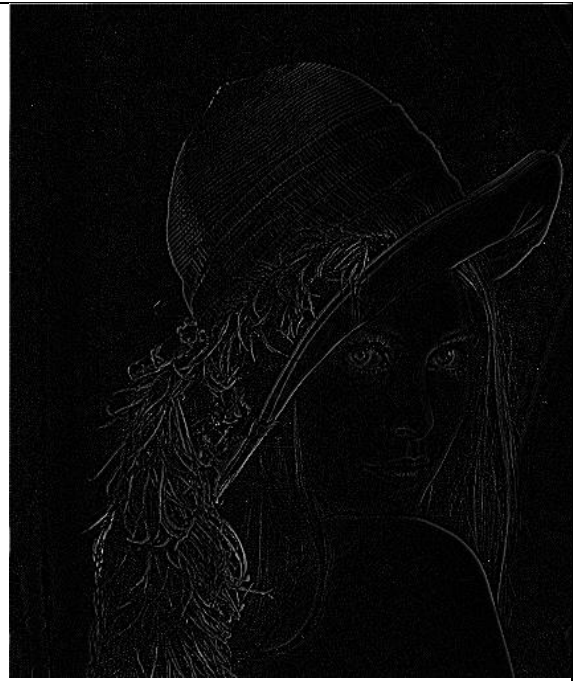
Original



Median Blur



Original



Edge Detection

REFERENCES

- MATLAB Documentation
- Library package documentations
 - NumPy
 - Matplotlib
 - OpenCV
- GeeksforGeeks
- Wikipedia articles
- The Medium
- Science Direct